A RAND NOTE

Why Is it Difficult to Program in von Neumann Languages?

Sanjai Narain

November 1987

RAND

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under RAND's National Defense Research Institute, a Federally Funded Research and Development Center supported by the Office of the Secretary of Defense, Contract No. MDA903-85-C-0030.

Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-2660-DARPA

Why Is it Difficult to Program in von Neumann Languages?

Sanjai Narain

November 1987

Prepared for
The Defense Advanced Research Projects Agency

RAND

PREFACE

This Note reports on research performed in the Knowledge-Based Simulation project. One of the main goals of this project is developing techniques for rapidly building large but correct computer models. Computer models are computer programs. Experience shows that it is far easier to develop programs in applicative languages than in von Neumann languages. If one could identify why this is true, one could obtain useful ideas for developing large but correct computer models. The Note should be useful to anyone with more than a cursory interest in computer science.

The research was conducted for the Information Science and Technology Office of the Defense Advanced Research Projects Agency (DARPA) under RAND's National Defense Research Institute, a Federally Funded Research and Development Center sponsored by the Office of the Secretary of Defense.

SUMMARY

Often the assertion is made that programs in von Neumann languages like Fortran, Pascal, or C are difficult to synthesize, analyze, modify, or extend. While the assertion appears to be correct, usual arguments given in its support are not very convincing. This Note focuses on onc part of this assertion, namely, that programs in von Neumann languages are difficult to synthesize, and develops a new argument in its support.

Frequently it is necessary to express in a programming language statements of the form "each of functions f1, . . . , fn are to be computed for the *same* entity." The presence of dcstructive assignmcnt in von Neumann languages gives rise to the possibility that programs modify their inputs. This makes it difficult to express statements of the above form in them. Applicative languages such as pure Lisp or pure Prolog do not contain destructive assignment, so such statements can be expressed very easily in them.

ACKNOWLEDGMENTS

I am grateful to D. Stott Parker, Norman Z. Shapiro, Jeff Rothenberg, and Christopher Burdorf for most helpful discussions on this Note.

CONTENTS

I. INTRODUCTION

Often the assertion is made that programs in von Neumann languages like Fortran, Algol, or C are difficult to synthesize, analyze, modify, or extend (e.g., Backus, 1978; Turner, 1982). Many arguments are given to support this assertion: Von Neumann languagcs do not have useful mathematical properties. Computation in these languages is thought of as incremental modification of some global statc, but this is a very primitive and unnatural way of thinking about it. Programs in these languages have to be mentally executed to be understood. The presence of destructive assignment in these languages causcs them to be referentially opaque, so it is not possible to substitute equals for equals, yet substitution of equals for equals is a very important rule in mathematics.

However, a number of objections can be raised against the above arguments. It is possible, in principle, to reason about a program in a von Neumann language in the traditional fashion using assertions and rules of inference (Hoare, 1969). Moreover, it is easy to associate a purely functional program with such a program (Henderson, 1979). To reason about the latter, why not just reason about the former? Even in the lambda calculus we think of computation in much the same way as in von Neumann languages. We think of a reduction as a computation, of each member in it as representing a state of computation, and of a reduction rule as transforming one state into another. There are many algorithms in computer science textbooks which, despite their dynamic nature or referential opacity, are not difficult to understand or reason about. The Turing machine is a von Neumann machine, yet we can prove very profound things about it.

Still, the original assertion does intuitively appear to be correct. This Note studies in dcpth one part of it, namely, that programs in von Neumann languages are difficult to synthesize, and develops a new argument in its support. The Note deals only with synthesis of programs for computing functions. However, a large number of problems in computer science can be thought of as problems of computing functions.

This Note proposes that development of a program to compute a function proceeds in two stages. In the first stage, an abstract algorithm for computing the function is developed. In the second stage, this is translated into a programming language. Substantial fragments of abstract algorithms are referentially transparent. In such a fragmcnt if there occur steps "compute f1 for x", . . . , "compute fn for x," where f1, . . , fn are functions, and each x has the same scope, then it is intended that each of f1, . . . , fn are to be computed for the *same* entity.

Von Neumann languages contain the destructive assignment operation. Its presence gives rise to the possibility that programs modify their inputs. This makes it difficult to translate into them the statement that each of f1, . . . , fn above are to be computed for the same entity. Since applicative languages such as pure Lisp (Henderson, 1979), pure Prolog (Lloyd, 1984), FP (Backus, 1978), and KRC (Turner, 1982) do not contain destructive assignment, it is straightforward to translate such statements into them.

II. COMPUTING FUNCTIONS WITH THE COMPUTER

A function is a mapping from a set of objects called its domain onto a set of objects called its range. To compute functions, algorithms are needed. An algorithm to compute a function is a systematic procedure such that by executing it a suitable agent could determine, in finite time, the value of the function for any given argument in its domain. An algorithm must also be effective, in that each step in it must be basic enough that a human can, in principle, perform it in a finite amount of time using only pencil and paper (Knuth, 1973).

Development of a program to compute a function can often be thought of as proceeding in two stages. In the first stage we develop an abstract algorithm for computing the function. This algorithm is usually in English or some other natural language, extended with appropriate mathematical and graphical notation. It is designed assuming the agent that will execute it is a human using pencil and paper. The structure it has and the ideas it contains are such that we can reason about it with other humans.

Examples of abstract algorithms for computing traditional functions can be found in mathematical textbooks (e.g., Kreyszig 1962). They are stated in English extended with mathematical notation and can be executed by a human using pencil and paper. In fact, most of them were invented before computers were, so it is natural that they were intended for execution by humans. They include algorithms for adding, multiplying, or dividing decimal numbers, for solving simultaneous linear equations for doing arithmetic on polynomials, or for solving problems in matrix algebra or in the differential and integral calculus.

Abstract algorithms are also developed for computing functions that typically arise in computer science. An example is the following from Aho et al. (1975), with minor syntactic modifications, for quicksorting a sequence of numbers S:

> To quicksort S check if S contains at most one element. If so return S. Otherwise choose an element 'a' randomly from S. Let S1, S2, and S3 be the sequences of elements in S less than, equal to, and greater than 'a', respectively. Return the result of quicksorting S1 followed by S2 followed by the result of quicksorting S3.

In the second stage, we try to implement or translate the abstract algorithm into a programming language. Let A be the abstract algorithm and L the programming language. Then, a program P in L is said to be an implementation of A if when the computer executes P, it simulates, at some appropriate level of abstraction, the actions of a human when he is executing A. Alternatively, if the behavior of P is paraphrased in English, A should result. For example, an implementation of the abstract quicksort algorithm in an Algol-like language is:

```
function quicksort(list:S):list;
 var a:integer;
 var S1,S2,S3: list;
 if length(S)=<1 then quicksort:=S else
    begin
       a:=middle_element(S);
       S1:=lesser_elements(a,S);
       S2:=equal_elements(a,S);
       S3:=greater_elements(a,S);
       quicksort:=append(quicksort(S1),S2,quicksort(S3))
    end
```

where lesser_elements is the name of a program computing that function which when applied to an element u and a sequence v returns all those elements of v less than u. Similarly, middle_element, length, equal_elements, greater_elements, and append are names of programs computing the obvious functions. It is clear that when the computer executes this program it simulates the behavior of a human executing the abstract quicksort algorithm.

A program P in L is said to be an implemcntation of an n-ary function f if, for any cntities e1, . . . , en, e, and their computer representations, respectively, E1, . . . , En, E, the following condition holds: f(e1, . . . , en) = e if and only if when E1, . . . , En are stored in the memory of the computcr and P executed, it halts with E storcd in the memory of the computer. Thus, if a program is an implementation of an abstract algorithm for computing a function, it is also an implementation of that function. However, the converse is not always truc.

For example, a program which implements the abstract quicksort algorithm also implements the sorting function. A program which implements the abstract mergesort algorithm also implements the sorting function. However, it cannot be said to implement the abstract quicksorting algorithm.

At this stage we also try to take advantage of the special capabilities of the programming language as well as work around its limitations. For example, we try to make as much use as possible of operations in it which are efficient. Usually, these include operations for arithmetic, logic, assignment, or random access to memory locations. We also try to be discreet about using operations in it which are slower. Usually, these include operations for disk access, recursive procedure calls, or dynamic memory allocation.

III. NATURE OF ABSTRACT ALGORITHMS

A fragment in an abstract algorithm is referentially transparent if within the scope of any name in the fragment, if the name has more than one occurrence, then at each occurrence it refers to the *same* entity. Note that a variable is a special kind of name. If in such a fragment there occur the steps "compute f1 for x", . . . , "compute fn for x," where each of f1, . . . , fn is a function, x is a variable, and each occurrence of x has the same scope, then, it is intended that each of f1, . . . , fn is to be computed for the same entity.

Most abstract algorithms for computing functions satisfy the following properties: (a) substantial fragments of them are referentially transparent; (b) inside such fragments, a variable frequently has more than one occurrence within its scope; (c) a predominant action in the algorithm is computation of a function.

The above properties can easily be verified for abstract algorithms in mathematical textbooks (e.g., Kreyszig, 1962). For example, consider Cramer's rule, an abstract algorithm for solving the simultaneous equations:

a1*x + b1*y = k1
a2*x + b2*y = k2

The rule is, let D be the determinant of [[a1,b1],[a2,b2]], D1 be the determinant of [[k1,b1],[k2,b2]], and D2 be the determinant of [[a1,k1],[a2,k2]]. The solution is x=D1/D and y=D2/D.

Here [x1, . . . , xn] represents a row x1, . . . , xn, and [R1, . . . , Rn] represents a matrix whose rows are represented by R1, . . . , Rn. The entire algorithm is referentially transparent. It is clear that at each occurrence, a1 refers to the same number. Similarly for a2, b1, b2, k1, k2, D1, D2, D. Also, actions of division and of obtaining determinants are computations of functions.

The above properties can also be verified for algorithms for computing functions that typically arise in computer science. This is not surprising, since such functions are not fundamentally different from those that arise in mathematics.

For example, consider again the abstract algorithm for sorting a sequence of numbers in Sec. II. The entire algorithm is referentially transparent. At each occurrence of variable S, S refers to the same sequence. At each occurrence, it is on this same sequence that we are to perform actions such as checking whether it contains at most one element, returning it, or obtaining elements less than a given element. All these actions are computations of functions.

Consider again, the abstract algorithm in Knuth (1973) for topologically sorting a set S on which a partial ordering =< has been imposed. The result is a sequence of all objects of S such that if x = <y, then x appears before y in the sequence. The algorithm, reproduced without modification, is:

> Take an object not preceded by any other object in S. This object may be placed first in the output. Now we remove this object from S. The resulting set is again partially ordered, and the process can be repeated until the whole set is sorted.

Again, the entire algorithm is referentially transparent. At both occurrences, S refers to the same set. At both occurrences, it is upon this same set that we are to perform the actions of selecting an object and of removing an object. Actions of selecting and deleting objects are also computations of functions.

IV. VON NEUMANN LANGUAGES

By a von Neumann language we mean a language that contains the destructive assignment operation. This operation makes it possible to overwrite contents of any arbitrary register in the memory of the computer. An example of it is the := operation of Algol. An instruction containing it is of the form A:=B, where A is the address of some register in the computer's memory and B is some expression. When it is executed, the value of B is stored in the register whose address is A. The previous contents of this register are lost.

This operation is a very useful one for several reasons. First, it can be performed extremely efficiently on present-day computers. Second, it can enable economical use of fast memory, which may be scarce. Information that is not needed can be overwritten with information that is. Third, it can reduce the need for dynamic memory allocation which has a certain overhead. For example, the abstract quicksort and heapsort algorithms can be implemented using arrays with destructively assignable cells in such a way that they use only a fixed amount of memory. This is a major reason for their importance. Fourth, it can be used for performing tasks requiring change to the physical world, e.g., in graphics or text editing.

However, of considerable importance also is the fact that this operation destroys information, i.e., information in the register upon which it is performed. This can lead to modification of objects in the computer memory in such a way that they cannot be recovered. For example, a sequence can be represented as a linked list of nodes. If the link field of the predecessor of a node is assigned to the address of its successor, then the node is effectively deleted. From the new list only, it is impossible to determine the original sequence.

In particular, it is entirely possible that a program in a von Neumann language may implement a function but modify the objects which are supplied to it as input. For example, in the quicksort and heapsort programs, a sequence is represented using an array of memory cells. The programs sort by rearranging contents of memory cells without requiring allocation of fresh cells. However, when they finish execution, the original sequence cannot be recovered. As another example, it is possible to implement integer division by 2 by a single right-shift on the register containing the number. However, after division, the original number is lost. It is this possibility—that programs modify their inputs—which makes programming in von Neumann languages difficult.

V. DIFFICULTY OF PROGRAMMING IN VON NEUMANN LANGUAGES

TRANSLATING REFERENTIALLY TRANSPARENT FRAGMENTS

Let A be an abstract algorithm and L a programming language into which we wish to translate A. Then it is likely that the algorithm contains a referentially transparent fragment c containing, in order, the steps "compute f1 for x", . . . , "compute fn for x," where f1, . . . , fn are functions, x is a variable, and each occurrence of x has the same scope. To translate A we have to translate c.

From what it means to implement A in L, the structure of the translation must be as close as possible to that of the abstract algorithm. Hence, the translation of c ought to be a fragment C in L satisfying two conditions: (a) C contains the calls F1(X), . . . , Fn(X) where X is a variable in L, and F1, . . . , Fn are implementations, respectively, of f1, . . . , fn; (b) when the computer is executing C, X contains the same object throughout C.

The difficulty which now arises is that one cannot let F1 be *any* implementation of f1, F2 be *any* implementation of f2, . . . , Fn be *any* implementation of fn. If we do let F1 be any implementation of f1, it is possible that it will compute f1 by modifying its input, i.e., the object in X, so the second condition upon C would not be satisfied. That is, F2, . . . , Fn would not be computed for the intended object.

For example, assuming that sequences are implemented as linked lists of nodes, the first node being the header and each successive node containing a key field and a next field, a reasonable definition of lesser_elements(x,S) in an Algol-like language is:

```
function lesser_elements(x:real; S:list):list;
 var A:list;
 begin
   A:=S;
   while not(null(A)) do
      if key(next(A))<x then A:=next(A);
      else next(A):=next(next(A));
   lesser_elements:=S
 end
```

Assuming that lists are passed by reference, this program proceeds by deleting from its input linked list all those nodes whose key field is equal to or greater than x. However, it cannot be used in the quicksort program. It modifies its input sequence in such a way that equal_elements will not compute the correct result.

SELECTING IMPLEMENTATIONS OF FUNCTIONS

Even though F1, . . . , Fn cannot be any implementations, respectively, of f1, . . . , fn, it may still be that selecting the correct implementations is not difficult. Unfortunately, this does not appear to be the case. Some obvious approaches are considered and it is shown that each poses serious problems.

First, one can let F1, . . . , Fn be those implementations, respectively, of f1, . . . , fn which do not modify their inputs. Then, provided no other steps in C modify X, X would be the same throughout C. However, given a program, determining whether it modifies its input can be quite nontrivial, particularly when it or the data structures it manipulates are large and complex. The problem is worse if pointers are being used or addresses are being computed at run-time. Of course, one could exercise the discipline of never using destructive assignment in one's own programs. But one cannot be sure that others do the same, so one cannot use their programs freely.

Second, one can let F1, . . . , Fn be arbitrary implementations, respectively, of f1, . . . , fn but resort to copying. That is, one could ensure that F1, . . . , Fn are always executed with copies of the object contained in X when C is entered, never with the original object itself. Then information about the object would never be lost. This approach is implicit in the call-by-value transmission of parameters to programs in von Neumann languages.

However, copying is feasible only when objects being manipulated require modest storage, such as numbers. It can be quite infeasible when these objects are arbitrary structures such as sequences, records, trees, or graphs. *In fact, copying can defeat one of the main purposes of using destructive assignment, which is to make economical use of memory.*

For example, in the quicksort program, if one were to precede every call to a program by copying its argument, the program would become quite wasteful of memory. Indeed one of the main advantages of quicksort, that it can be implemented in a fixed amount of memory, would be lost.

Third, we could let F1, . . . , Fn be programs which do modify their inputs but in a collectively benign fashion, that is, satisfying the following safety condition:

> For every i the information that Fi destroys is not needed by any other Fj, j>i. That is, place any object E in the domain of F1, . . . , Fn in X and let Fi execute. Then, if Fj is executed with the object remaining in X, it yields as output the same object which Fj would yield as output, had it been executed with E in X.

Then, even though the second condition upon C would be violated, the objects computed by F1, . . . , Fn would still be the same as when X contains the same object throughout C. Moreover, the structure of C would remain close to that of c and one would also avoid copying. For example, instead of deleting nodes whose key field is equal to or greater than x, lesser_elements could delete nodes whose key field is less than x and store them in a separate list. Definitions of equal_elements and greater_elements could be similar. The definition of quicksort would then remain unchanged.

However, there still exists a major problem. In general, it is very difficult to determine what combination of implementations of f1, . . . , fn satisfies the above safety condition. If there are at least k implementations each of f1, . . . , fn, then there are, in the worst case, n*k combinations to check. Given a combination, there are, in the worst case, n*(n–1)/2 interactions to check. The total worst case complexity is then o(k*n^3). Even checking whether the information that a program destroys is not needed in later programs is nontrivial.

VI. APPLICATIVE LANGUAGES

PROGRAMMING IN APPLICATIVE LANGUAGES

Applicative languages are based on various calculi of mathematical logic, notably the lambda calculus and the calculus of Horn clauses with SLD-resolution (Lloyd, 1984). The term "applicative" derives from the fact that one of the main operations in these languages is that of function application. A program in these languages is an expression in one of these calculi. When interpreters in these calculi, such as applicative-order reduction or SLD-resolution, interpret or execute a program, they simulate the execution of some abstract algorithm by the human.

One of the most important features of applicative languages which distinguishes them from von Neumann languages is that they do not contain the destructive assignment operation. This is not surprising, since they are based on mathematical logic which does not contain this operation. Examples of applicative languages are pure Lisp, FP, KRC, and pure Prolog.

Since destructive assignment is absent in applicative languages it is impossible to write a program which modifies its input. Hence, in Sec. V we can let F1, . . . , Fn be *any* implementations, respectively, of f1, . . . , fn. This constitutes a drastic simplification over the von Neumann case and accounts in a major way for the ease of programming in applicative languages.

LIMITATIONS OF APPLICATIVE LANGUAGES

The lack of destructive assignment does limit the power of applicative languages. One can no longer make local changes to large objects in computer memory even when one is sure that the information lost will never be needed. For example, one cannot implement arrays in applicative languages with constant access and update time (Henderson, 1979). As a consequence, one cannot efficiently implement graph algorithms.

In fact, computation in applicative languages is thought of not as successive modification of some initial state, but as building new structures from old. For example, sorting a list is thought of as building a new list which is the sorted version of the old list. Thus computations can involve considerable copying of information and, consequently, considerable dynamic memory allocation. So, applicative programs can be substantially less efficient than equivalent von Neumann programs executing in a fixed amount of memory.

If memory is in short supply, then memory management techniques such as garbage collection or cdr-coding (Weinreb and Moon, 1981) are used. Their use has a certain overhead and can further slow down the execution of applicative programs.

However, copying in applicative languages is quite different from the indiscriminate copying of the second approach in Sec. V, in which all arguments are copied upon procedure entry. In the former, it can be controlled by good algorithm design, whereas in the latter it cannot be. For example, in pure Lisp, memory cells are allocated only by a call to cons, so in the following program:

```
(defun append (x y)
   (if (null x) then y else (cons (car x) (append (cdr x) y))))
```

the number of cells allocated is equal only to the length of x. With the second approach of Sec. V, the number of cells allocated would be equal to the sum of the lengths of x and y.

VII. SUMMARY AND DISCUSSION

This Note studies the assertion that programs in von Neumann languages are difficult to synthesize and develops a new argument in its support. Frequently it is necessary to express in a programming language statements of the form "each of functions f1, . . . , fn are to be computed for the *same* entity." Since von Neumann languages contain the destructive assignment operation, it is very difficult to express such statements in them. Since applicative languages do not contain this operation, such statements can be expressed very easily in them, however, sometimes at the loss of some efficiency.

This Note has attempted to go beyond merely criticizing destructive assignment as mathematically inelegant and to better understand its role in programming. Making economical use of computer memory is an important goal, and, given present computer architectures, destructive assignment is a powerful tool for achieving it. Moreover, it is an extremely fast operation, which is another good reason to consider its use. However, its use raises a certain difficulty.

Note that a von Neumann language is defined as any language containing the destructive assignment operation. Thus, the discussion of this Note also applies to full Lisp systems such as that of Weinreb and Moon (1981) which contain operations such as setq or rplaca. Thus, the very presence of this operation in a language will raise the above difficulty. It does not seem possible to resolve this difficulty in general, though of course, in special cases, it may be resolved quite easily.

Another, less direct appreciation of this difficulty can be obtained by considering the following curious and unintuitive situation. Programs may implement functions, but if they contain destructive assignment, they *cannot* be manipulated in ways that functions can be. For example, even if we notice that for some functions h,r,f,g we have h(x) = r(f(x),g(x)) and that programs R,F,G implement respectively r,f,g, we cannot assume that H implements h by defining H(X) = R(F(X),G(X)). F may modify contents of X.

In general, then, it is not possible to use von Neumann language implementations of functions in contexts not intended or imagined when they were developed. We cannot implement a function once and for all and use it without modification in the implementation of any abstract algorithm in which the function appears. This partially explains the difficulty of *software reuse* in von Neumann languages.

Memory is already abundantly available, and it is getting cheaper. Techniques such as tail-recursion compilation further optimize memory utilization of applicative programs. Parallel architectures are expected to support applicative languages more efficiently than sequential ones. There are many applications for which it is not critical to write the most space- and time-efficient programs. Given these facts, applicative languages are and will become very practical.

Finally, we can add a word about the difficulty of modifying von Neumann programs. Suppose we choose the third approach in Sec. V. If for some reason we need to associate another program with Fi, we cannot freely choose any other implementation of fi. The program currently associated with Fi does not destroy information necessary for correct execution of Fj, j>i, but it is quite possible that a replacement would. Of course, this problem does not arise in applicative languages.

REFERENCES

Abelson, H., and Sussman, G. [1984]. Structure and interpretation of computer programs. MIT Press and McGraw Hill, Cambridge, MA.

Aho, A., Hopcroft, J., and Ullman, J. [1974]. The design and analysis of computer algorithms. Addison-Wesley Publishing Company, Reading, MA.

Backus, J. [1978]. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM August 1978.

Dijkstra, E. [1976]. A discipline of programming. Prentice Hall, Englewood Cliffs, NJ.

Henderson, P. [1979]. Functional programming: application and implementation. Prentice-Hall, Englewood Cliffs, NJ.

Hoare, C. [1969]. An axiomatic basis for computer programming. CACM October 1969.

Knuth, D. [1973]. The art of computer programming. Addison-Wesley Publishing Company, Reading, MA.

Kreyszig, E. [1962]. Advanced engineering mathematics. John Wiley and Sons, New York, NY.

Lloyd, J. [1984]. Foundations of logic programming. Springer Verlag, New York, NY.

Scott, D., and Scherlis, W. [1983]. Towards inferential programming. Proceedings, IFIP Congress.

Simon, H. [1969]. The sciences of the artificial. MIT Press, Cambridge, MA.

Turner, D. A. [1982]. Recursion equations as a programming language. In Functional programming and its applications: An advanced course, J. Darlington, P. Henderson, and D. A. Turner (eds.). Cambridge University Press, New York, NY.

Weinreb, D., and Moon, D. [1981]. Lisp machine manual. Massachusetts Institute of Technology, Cambridge, MA.

RAND/N-2660-DARPA